Pandas Essentials

Introduction to Pandas

Pandas (https://pandas.pydata.org) is an open source library providing high-performance, easy-to-use data structures and data analysis tools for the Python programming language. Today, pandas is actively supported by a community of like-minded individuals around the world who contribute their valuable time and energy to help make open source pandas possible.

Python for Data Analysis (http://shop.oreilly.com/product/0636920023784.do) is a great read by **Wes McKinney** who is the **creator of Pandas library**.

In this section of the course, we will learn to use pandas for data analysis. If you have never used pandas, you can think about pandas as an extremely powerful version of Excel and with lot more features.
We will cover the following key concepts in this section (Pandas Essentials) of the course:

- Series
- DataFrame
- Indexing and Selection
- Hierarchical Indexing
- Data Cleaning, Preparation and Handling the Missing Data
- Data Wrangling: Merging/Joining, Combining/Concatenation
- Data Aggregation and GroupBy

Several other Useful Methods and Operations and much more, and at the end two full data analysis exercises to practice the skills.

pandas Data Structures:

Series and **DataFrame** are **two workhorse** data structures in pandas.
Lets talk about series first:

Series:

Series is a one-dimensional array-like object, which contains values and an array of labels, associated with the values. Series can be indexed using labels.What differentiates the NumPy array from a Series, is that a Series can have axis labels, meaning it can be indexed by a label, instead of just a number location.(Series is similar to NumPy array -- actually, it is built on top of the NumPy array object).Series can hold any arbitrary Python object.

It has to be remembered that unlike Python lists, a Series will always contain data of the same type.

In [1]:
```
# first thing first, we need to import NumPy and pandas
# np and pd are alias for NumPy and pandas
# pandas Documentation using pd.<TAB> and pd?
import numpy as np
import pandas as pd
```

We can create a Series using list, numpy array, or dictionary
Let's create these objects and convert them into panda's Series!

1-Series using lists
Lets create a Python list containing labels and another with data.

In [2]:
```
my_labels = ['x','y','z']
my_data = [100,200,300]
```

So, we have two Python's list objects,

- `my_labels` - a list of strings, and
- `my_data` - a list of numbers

We can use **`pd.Series`** (with capital S) to convert the Python's list object to pandas Series.

☞ *If you press `<Shift+tab>` , you see Series takes a wide variety of parameters, at the moment we will focus on the data and the index. lets consider data only and see how it works!*

In [3]:
```
# Converting my_data (Python list) to Series (pandas series)
pd.Series(data = my_data)
# shift+tab, we will focus on data and index at the moment
```

Out[3]:
```
0    100
1    200
2    300
dtype: int64
```

Column "0 1 2" is automatically generated index for the elements in series with data 100 200 and 300. We can specify index values and call the data points using these indexes.
Let's pass "my_labels" to the Series as index.

In [4]:
```
pd.Series(data = my_data, index = my_labels)
#pd.Series(my_data, my_labels) # Becasue data and index are in order (shit+T
```

Out[4]:
```
x    100
y    200
z    300
dtype: int64
```

2-Series using NumPy arrays

In [5]:
```
# Lets create NumPy array from my_data and then Series from that array
my_array = np.array(my_data)
pd.Series(data = my_array)
```

Out[5]:
```
0    100
1    200
2    300
dtype: int32
```

In [6]:
```
pd.Series(data = my_array, index = my_labels)
# pd.Series(my_array, my_labels) # data and index are in order
```

Out[6]:
```
x    100
y    200
z    300
dtype: int32
```

3-Series using dictionary

In [7]:
```
# Let's create a dictionary my_dic
my_dic = {'x':100,'y':200,'z':300}
pd.Series(my_dic)
```

Out[7]:
```
x    100
y    200
z    300
dtype: int64
```

Notice the difference here,
if we pass a `dictionary` to `Series` , pandas will take the keys as `index/labels` and `values` as data.

Series can hold a wide variety of objects types, lets see with examples:

In [8]:
```
# let's pass my_labels (which is a list of strings) as data
pd.Series(data = my_labels)
```

Out[8]:
```
0    x
1    y
2    z
dtype: object
```

In [9]:
```
# We can pass a list of buit-in functions!
pd.Series([min, max, sum, print])
# This is jsut an example, you may not see this in the real world!
```

Out[9]:
```
0      <built-in function min>
1      <built-in function max>
2      <built-in function sum>
3    <built-in function print>
dtype: object
```

Grabbing data from Series:

Indexes are the key thing to understand in Series. Pandas use these indexes (numbers or names) for fast information retrieval. (Index works just like a hash table or a dictionary).

To understand the concepts, Let's create three Series, `ser1` , `ser2` , `ser3` from dictionaries with some random data:

In [10]:
```
# Creating dictionaries
dic_1 = {'Toronto': 500, 'Calgary': 200, 'Vancouver': 300, 'Montreal': 700}
dic_2 = {'Calgary': 200, 'Vancouver': 300, 'Montreal': 700}
dic_3 = {'Calgary': 200, 'Vancouver': 300, 'Montreal': 700, 'Jasper':1000}
```

In [11]:
```
# Creating pandas series from the dictionaries
ser1 = pd.Series(dic_1)
ser2 = pd.Series(dic_2)
ser3 = pd.Series(dic_3)
```

In [12]:
```
# Grabbing information for series is very much similar to dictionary.
ser1['Calgary'] # its case sensitive "calgary" is not the same as "Calgary"
```

Out[12]:
```
200
```

Note, we are passing a string "Calgary" our index contains strings (name of the cities). If the index is a number, we will pass in the number.

In [13]:
```
ser1 # Order of key is same that what is given in the dictonary
```

Out[13]:
```
Toronto      500
Calgary      200
Vancouver    300
Montreal     700
dtype: int64
```

In [14]:
```
ser2
```

Out[14]:
```
Calgary      200
Vancouver    300
Montreal     700
dtype: int64
```

Basic operations on series are usually based on the index.
For example, if we want to add ser1 + ser2, it tries to match up the operation based on the index. For Calgary, Montreal and Vancouver, it adds the values whereas for Toronto, it can not find a match and put NaN there.

In [15]:
```
ser4 = ser1 + ser2
ser4  # Key will appear as Alphabetical Order
```

Out[15]:
```
Calgary       400.0
Montreal     1400.0
Toronto         NaN
Vancouver     600.0
dtype: float64
```

In [16]:
```
# Let's look at ser3!
ser3
```

Out[16]:
```
Calgary       200
Vancouver     300
Montreal      700
Jasper       1000
dtype: int64
```

In [17]:
```
ser5 = ser4 + ser3
ser5
```

Out[17]:
```
Calgary       600.0
Jasper          NaN
Montreal     2100.0
Toronto         NaN
Vancouver     900.0
dtype: float64
```

Notice that the values found in the series were added for their appropriate index, on the other hand, if there is no match, the value appears as NaN (not a number) which is considered in pandas to **mark missing or NA values**.

Good to know!

`isnull()`, `notnull()`

- detect missing data

In [18]:
```
#pd.isnull(ser4)
ser4.isnull()
# shift+tab, its Type is method
```

Out[18]:
```
Calgary      False
Montreal     False
Toronto       True
Vancouver    False
dtype: bool
```

In [19]:
```
#pd.notnull(ser5)
ser4.notnull()
```

Out[19]:
```
Calgary       True
Montreal      True
Toronto      False
Vancouver     True
dtype: bool
```

axes , values

- `axes` : returns list of the row axis labels/index
- `values` : returns list of values/data

Let's try `axes` and `values` on our series!

In [20]:
```
# row axis labels (index) list can be obtained
ser1.axes
#<shift+tab> axes type is property, its attribute!
```

Out[20]:
```
[Index(['Toronto', 'Calgary', 'Vancouver', 'Montreal'], dtype='object')]
```

In [21]:
```
# returns the values/data
ser1.values
```

Out[21]:
```
array([500, 200, 300, 700], dtype=int64)
```

head() , tail()

To view a small sample of a Series or DataFrame (we will learn `DataFrame` in the next lecture) object, use the **`head()`** and **`tail()`** methods.
The default number of elements to display is **five**, but you may pass a custom number.

In [22]:
```
ser1.head(1)
```

Out[22]:
```
Toronto    500
dtype: int64
```

In [23]:
```
ser1.tail(1)
```

Out[23]:
```
Montreal    700
dtype: int64
```

size

- To check the number of elements in your data.

In [24]:
```
1 ser1.size
```

Out[24]: 4

empty

- True if the series in empty

In [25]:
```
1 # True for empty series
2 ser1.empty
```

Out[25]: False

pandas Data Structures:

We have learned about **Series**, lets learn DataFrames (2nd workhorse of pandas) to expand our concepts of Series.

DataFrame

- A very simple way to think about the DataFrame is, "bunch of Series together such as they share the same index".
- A DataFrams is a rectangular table of data that contains an ordered collection of columns, each of which can be a different value type (numeric, string, boolean, etc). DataFrame has both a row and column index; it can be thought of as a dictionary of Series all sharing the same index.
- A DataFrams can be created by following methods-

```
  * Passing Data into Data frame function as Numpy array with index
 for row and column as sepaerate List(By Default
    number index strating with 0,1,2.... in both cases).
  * Passing Data into Data frame function as list of values as colum
ns using Zip function with index for row and
    column as sepaerate List (By Default number index strating with
 0,1,2.... in both cases).
  * Another way is to Passing Data into Data frame function as dicti
onary , you will not need to supply column names
    separately in this case , However you can pass index for rows (B
y Default row index will be numbers 0,1,2...).
  * Reading the data from file using Pandas package functions.
  * just the process of creating them is different , there is no dif
ference in properties of end result.
```

☞ *A good read for those, who are interested! (Python for Data Analysis (http://shop.oreilly.com/product/0636920023784.do))*

Let's learn **DataFrame with examples:**

Method-1

Let's create two labels/indexes:

- for rows 'r1 to r10'
- for columns 'c1 to c10'

Let's start with a simple example, using **`arange()`** and **`reshape()`** together to create a 2D array (matrix).

In [26]:
```
index = 'r1 r2 r3 r4 r5 r6 r7 r8 r9 r10'.split()
columns = 'c1 c2 c3 c4 c5 c6 c7 c8 c9 c10'.split()
array_2d = np.arange(0,100).reshape(10,10)
```

✅ *Use **TAB** for auto-complete and **shift + TAB** for doc.*

In [27]:
```
# How the index, columns and array_2d look like!
index
```

Out[27]:
```
['r1', 'r2', 'r3', 'r4', 'r5', 'r6', 'r7', 'r8', 'r9', 'r10']
```

In [28]:
```
columns
```

Out[28]:
```
['c1', 'c2', 'c3', 'c4', 'c5', 'c6', 'c7', 'c8', 'c9', 'c10']
```

In [29]:
```
array_2d
```

Out[29]:
```
array([[ 0,  1,  2,  3,  4,  5,  6,  7,  8,  9],
       [10, 11, 12, 13, 14, 15, 16, 17, 18, 19],
       [20, 21, 22, 23, 24, 25, 26, 27, 28, 29],
       [30, 31, 32, 33, 34, 35, 36, 37, 38, 39],
       [40, 41, 42, 43, 44, 45, 46, 47, 48, 49],
       [50, 51, 52, 53, 54, 55, 56, 57, 58, 59],
       [60, 61, 62, 63, 64, 65, 66, 67, 68, 69],
       [70, 71, 72, 73, 74, 75, 76, 77, 78, 79],
       [80, 81, 82, 83, 84, 85, 86, 87, 88, 89],
       [90, 91, 92, 93, 94, 95, 96, 97, 98, 99]])
```

In [30]:
```
# Let's create our first DataFrame using index, columns and array_2dnow
df = pd.DataFrame(data = array_2d, index = index, columns = columns)
```

In [31]:
```
# How the DataFrame look like!
df
```

Out[31]:

	c1	c2	c3	c4	c5	c6	c7	c8	c9	c10
r1	0	1	2	3	4	5	6	7	8	9
r2	10	11	12	13	14	15	16	17	18	19
r3	20	21	22	23	24	25	26	27	28	29
r4	30	31	32	33	34	35	36	37	38	39
r5	40	41	42	43	44	45	46	47	48	49
r6	50	51	52	53	54	55	56	57	58	59
r7	60	61	62	63	64	65	66	67	68	69
r8	70	71	72	73	74	75	76	77	78	79
r9	80	81	82	83	84	85	86	87	88	89
r10	90	91	92	93	94	95	96	97	98	99

Method-2

In [32]:
```
age=np.random.randint(low=16,high=80,size=[20,])
city=np.random.choice(['Mumbai','Delhi','Chennai','Kolkata'],20)
default=np.random.choice([0,1],20)
```

In [33]:
```
age
```

Out[33]:
```
array([57, 30, 22, 53, 60, 19, 32, 35, 17, 37, 44, 71, 72, 47, 66, 23, 75,
       37, 20, 23])
```

In [34]:
```
city
```

Out[34]:
```
array(['Delhi', 'Chennai', 'Delhi', 'Chennai', 'Kolkata', 'Delhi',
       'Delhi', 'Chennai', 'Chennai', 'Mumbai', 'Delhi', 'Delhi',
       'Kolkata', 'Mumbai', 'Delhi', 'Kolkata', 'Delhi', 'Chennai',
       'Chennai', 'Delhi'], dtype='<U7')
```

In [35]:
```
default
```

Out[35]:
```
array([0, 0, 0, 1, 0, 1, 1, 1, 0, 0, 1, 1, 1, 0, 1, 0, 1, 0, 0, 1])
```

we can zip these to convert them to single list of tuples , each tuple in that list will correspond to a row in the dataframe

In [36]:
```
mydata=list(zip(age,city,default))
```

In [37]:
```
1 mydata
```

Out[37]:
```
[(57, 'Delhi', 0),
 (30, 'Chennai', 0),
 (22, 'Delhi', 0),
 (53, 'Chennai', 1),
 (60, 'Kolkata', 0),
 (19, 'Delhi', 1),
 (32, 'Delhi', 1),
 (35, 'Chennai', 1),
 (17, 'Chennai', 0),
 (37, 'Mumbai', 0),
 (44, 'Delhi', 1),
 (71, 'Delhi', 1),
 (72, 'Kolkata', 1),
 (47, 'Mumbai', 0),
 (66, 'Delhi', 1),
 (23, 'Kolkata', 0),
 (75, 'Delhi', 1),
 (37, 'Chennai', 0),
 (20, 'Chennai', 0),
 (23, 'Delhi', 1)]
```

In [38]:
```
1 df1=pd.DataFrame(data=mydata,columns=['age','city','default'])
```

In [39]:
```
1 df1
```

Out[39]:

	age	city	default
0	57	Delhi	0
1	30	Chennai	0
2	22	Delhi	0
3	53	Chennai	1
4	60	Kolkata	0
5	19	Delhi	1
6	32	Delhi	1
7	35	Chennai	1
8	17	Chennai	0
9	37	Mumbai	0
10	44	Delhi	1
11	71	Delhi	1
12	72	Kolkata	1
13	47	Mumbai	0
14	66	Delhi	1
15	23	Kolkata	0
16	75	Delhi	1
17	37	Chennai	0
18	20	Chennai	0
19	23	Delhi	1

Method-3

Another way is to put them in a dictionary , you will not need to supply column names separately in this case

In [40]:
```
1 df2=pd.DataFrame({'age':age,'city':city,'default':default})
```

In [41]:
```
1 df2
```

Out[41]:

	age	city	default
0	57	Delhi	0
1	30	Chennai	0
2	22	Delhi	0
3	53	Chennai	1
4	60	Kolkata	0
5	19	Delhi	1
6	32	Delhi	1
7	35	Chennai	1
8	17	Chennai	0
9	37	Mumbai	0
10	44	Delhi	1
11	71	Delhi	1
12	72	Kolkata	1
13	47	Mumbai	0
14	66	Delhi	1
15	23	Kolkata	0
16	75	Delhi	1
17	37	Chennai	0
18	20	Chennai	0
19	23	Delhi	1

df is our first dataframe.
We have columns, c1 to c10, and their corresponding rows, r1 to r10.
Each column is actually a pandas series, sharing a common index (row labels).

☞ Let's learn how to **Grab data** that we need, this is the most important thing we want to learn to move one!

Columns

In [42]:
```
# Grabbing a single column
df['c1']
# The output looks like a series, right?.
# Also returned Series have the same index as the DataFrame
```

Out[42]:
```
r1      0
r2     10
r3     20
r4     30
r5     40
r6     50
r7     60
r8     70
r9     80
r10    90
Name: c1, dtype: int32
```

In [43]:
```
type(df['c1']) # It is a pandas Series
```

Out[43]:
```
pandas.core.series.Series
```

In [44]:
```
# Grabbing more than one column, pass the list of columns you need!
df[['c1', 'c10']]
```

Out[44]:

	c1	c10
r1	0	9
r2	10	19
r3	20	29
r4	30	39
r5	40	49
r6	50	59
r7	60	69
r8	70	79
r9	80	89
r10	90	99

df.column_name (e.g. df.c1, df.c2 etc) can be used to grab a column as well, its good to know but I don't recommend.
If you press "TAB" after df., you will see lots of available methods, its good not to get confused with these option by using df.column_name.
Let's try this once

In [45]:
```
1 df.c5
```

Out[45]:
```
r1      4
r2     14
r3     24
r4     34
r5     44
r6     54
r7     64
r8     74
r9     84
r10    94
Name: c5, dtype: int32
```

Adding new column

Lets try with "+" operation!

In [46]:
```
1 df['new'] = df['c1'] + df['c2']
2 df
```

Out[46]:

	c1	c2	c3	c4	c5	c6	c7	c8	c9	c10	new
r1	0	1	2	3	4	5	6	7	8	9	1
r2	10	11	12	13	14	15	16	17	18	19	21
r3	20	21	22	23	24	25	26	27	28	29	41
r4	30	31	32	33	34	35	36	37	38	39	61
r5	40	41	42	43	44	45	46	47	48	49	81
r6	50	51	52	53	54	55	56	57	58	59	101
r7	60	61	62	63	64	65	66	67	68	69	121
r8	70	71	72	73	74	75	76	77	78	79	141
r9	80	81	82	83	84	85	86	87	88	89	161
r10	90	91	92	93	94	95	96	97	98	99	181

Deleting the column -- `drop()`

```
*df.drop('new')-- ValueError: labels ['new'] not contained in axis
```

Shift+tab, you see the default axis is 0, which refers to the index (row labels), for column, we need to specify axis = 1.

☞ rows refer to 0 axis and columns refers to 1 axis

☞ Quick Check: *df.shape gives tuple (rows, cols) at [0] and [1]*

In [47]:
```
# We can delete a column using drop()
# df.drop('new')# ValueError: labels ['new'] not contained in axis
df.drop('new', axis=1)
```

Out[47]:

	c1	c2	c3	c4	c5	c6	c7	c8	c9	c10
r1	0	1	2	3	4	5	6	7	8	9
r2	10	11	12	13	14	15	16	17	18	19
r3	20	21	22	23	24	25	26	27	28	29
r4	30	31	32	33	34	35	36	37	38	39
r5	40	41	42	43	44	45	46	47	48	49
r6	50	51	52	53	54	55	56	57	58	59
r7	60	61	62	63	64	65	66	67	68	69
r8	70	71	72	73	74	75	76	77	78	79
r9	80	81	82	83	84	85	86	87	88	89
r10	90	91	92	93	94	95	96	97	98	99

☞ Is the "new" really deleted?
Output df and you will see "new" is still there!

In [48]:
```
df
```

Out[48]:

	c1	c2	c3	c4	c5	c6	c7	c8	c9	c10	new
r1	0	1	2	3	4	5	6	7	8	9	1
r2	10	11	12	13	14	15	16	17	18	19	21
r3	20	21	22	23	24	25	26	27	28	29	41
r4	30	31	32	33	34	35	36	37	38	39	61
r5	40	41	42	43	44	45	46	47	48	49	81
r6	50	51	52	53	54	55	56	57	58	59	101
r7	60	61	62	63	64	65	66	67	68	69	121
r8	70	71	72	73	74	75	76	77	78	79	141
r9	80	81	82	83	84	85	86	87	88	89	161
r10	90	91	92	93	94	95	96	97	98	99	181

To delete the column, you have to tell the pandas by setting

- ***inplace = True*** (default is inplace=False).

✅ *pandas is generous, it does not want us to lose the information by any mistake and needs inplace*

In [49]:
```
df.drop('new',axis = 1, inplace = True)
df
```

Out[49]:

	c1	c2	c3	c4	c5	c6	c7	c8	c9	c10
r1	0	1	2	3	4	5	6	7	8	9
r2	10	11	12	13	14	15	16	17	18	19
r3	20	21	22	23	24	25	26	27	28	29
r4	30	31	32	33	34	35	36	37	38	39
r5	40	41	42	43	44	45	46	47	48	49
r6	50	51	52	53	54	55	56	57	58	59
r7	60	61	62	63	64	65	66	67	68	69
r8	70	71	72	73	74	75	76	77	78	79
r9	80	81	82	83	84	85	86	87	88	89
r10	90	91	92	93	94	95	96	97	98	99

Rows

We can retrieve a row by its name or position with **`loc`** (https://pandas.pydata.org/pandas-docs/stable/generated/pandas.DataFrame.loc.html) and **`iloc`** (https://pandas.pydata.org/pandas-docs/stable/generated/pandas.DataFrame.iloc.html).
loc -- Access a group of rows and columns by label(s)

In [50]:
```
# df['r1'] # KeyError: 'r1'
df.loc['r1'] # loc for location in square brackets
# we see that the rows are series as well!
```

Out[50]:
```
c1     0
c2     1
c3     2
c4     3
c5     4
c6     5
c7     6
c8     7
c9     8
c10    9
Name: r1, dtype: int32
```

In [51]:
```
type(df.loc['r1'])
```

Out[51]:
```
pandas.core.series.Series
```

In [52]:
```
ser1=df.loc['r1']
ser1['c1']
#   same as above 2 lines df.loc['r1']['c1']
```

Out[52]:
```
0
```

Using row's index location with **iloc**, even if our index is labeled.

In [53]:
```
df.iloc[0] # iloc[index], index based location
```

Out[53]:
```
c1     0
c2     1
c3     2
c4     3
c5     4
c6     5
c7     6
c8     7
c9     8
c10    9
Name: r1, dtype: int32
```

In [54]:
```
# more than one rows -- pass a list of rows!
df.loc[['r1','r2']]
```

Out[54]:

	c1	c2	c3	c4	c5	c6	c7	c8	c9	c10
r1	0	1	2	3	4	5	6	7	8	9
r2	10	11	12	13	14	15	16	17	18	19

Grabbing an element or a sub-set of the dataframe

In [55]:
```
# df.loc(req_row, re_col) -- pass row, col for the element!
df.loc['r1','c1']
```

Out[55]:
```
0
```

In [56]:
```
# for a sub-set, pass the list
df.loc[['r1','r2'],['c1','c2']]
```

Out[56]:

	c1	c2
r1	0	1
r2	10	11

In [57]:
```
# another example - random columns and rows in the list
df.loc[['r2','r5'],['c3','c4']]
```

Out[57]:

	c3	c4
r2	12	13
r5	42	43

In [58]:
```
# one more example - random columns and rows in the list
df.loc[['r2','r5'],['c3','c4','c5']]
```

Out[58]:

	c3	c4	c5
r2	12	13	14
r5	42	43	44

In [59]:
```
# We can do a conditional selection as well
df > 5

# df!=0
# df=0
```

Out[59]:

	c1	c2	c3	c4	c5	c6	c7	c8	c9	c10
r1	False	False	False	False	False	False	True	True	True	True
r2	True	True	True	True	True	True	True	True	True	True
r3	True	True	True	True	True	True	True	True	True	True
r4	True	True	True	True	True	True	True	True	True	True
r5	True	True	True	True	True	True	True	True	True	True
r6	True	True	True	True	True	True	True	True	True	True
r7	True	True	True	True	True	True	True	True	True	True
r8	True	True	True	True	True	True	True	True	True	True
r9	True	True	True	True	True	True	True	True	True	True
r10	True	True	True	True	True	True	True	True	True	True

In [60]:
```
1 df[df>5]
```

Out[60]:

	c1	c2	c3	c4	c5	c6	c7	c8	c9	c10
r1	NaN	NaN	NaN	NaN	NaN	NaN	6	7	8	9
r2	10.0	11.0	12.0	13.0	14.0	15.0	16	17	18	19
r3	20.0	21.0	22.0	23.0	24.0	25.0	26	27	28	29
r4	30.0	31.0	32.0	33.0	34.0	35.0	36	37	38	39
r5	40.0	41.0	42.0	43.0	44.0	45.0	46	47	48	49
r6	50.0	51.0	52.0	53.0	54.0	55.0	56	57	58	59
r7	60.0	61.0	62.0	63.0	64.0	65.0	66	67	68	69
r8	70.0	71.0	72.0	73.0	74.0	75.0	76	77	78	79
r9	80.0	81.0	82.0	83.0	84.0	85.0	86	87	88	89
r10	90.0	91.0	92.0	93.0	94.0	95.0	96	97	98	99

This is similar to NumPy boolean mask, lets try this:

```
*bool_mask = df % 3 == 0
*df[bool_mask]
```

returns values where it is True and NaN where False.

In [61]:
```
1 # Return Divisible by 3
2 bool_mask = df % 3 == 0
3 df[bool_mask]
4 # One step and easier to do
5 # df[df % 3 == 0]
```

Out[61]:

	c1	c2	c3	c4	c5	c6	c7	c8	c9	c10
r1	0.0	NaN	NaN	3.0	NaN	NaN	6.0	NaN	NaN	9.0
r2	NaN	NaN	12.0	NaN	NaN	15.0	NaN	NaN	18.0	NaN
r3	NaN	21.0	NaN	NaN	24.0	NaN	NaN	27.0	NaN	NaN
r4	30.0	NaN	NaN	33.0	NaN	NaN	36.0	NaN	NaN	39.0
r5	NaN	NaN	42.0	NaN	NaN	45.0	NaN	NaN	48.0	NaN
r6	NaN	51.0	NaN	NaN	54.0	NaN	NaN	57.0	NaN	NaN
r7	60.0	NaN	NaN	63.0	NaN	NaN	66.0	NaN	NaN	69.0
r8	NaN	NaN	72.0	NaN	NaN	75.0	NaN	NaN	78.0	NaN
r9	NaN	81.0	NaN	NaN	84.0	NaN	NaN	87.0	NaN	NaN
r10	90.0	NaN	NaN	93.0	NaN	NaN	96.0	NaN	NaN	99.0

☞ Its not common to use such operation on entire dataframe. We usually use them on a columns

or rows instead.
For example, we don't want a row with NaN values.
What to do?
Let's have a look at one example.

In [62]:
```
# Our original dataframe is
df
```

Out[62]:

	c1	c2	c3	c4	c5	c6	c7	c8	c9	c10
r1	0	1	2	3	4	5	6	7	8	9
r2	10	11	12	13	14	15	16	17	18	19
r3	20	21	22	23	24	25	26	27	28	29
r4	30	31	32	33	34	35	36	37	38	39
r5	40	41	42	43	44	45	46	47	48	49
r6	50	51	52	53	54	55	56	57	58	59
r7	60	61	62	63	64	65	66	67	68	69
r8	70	71	72	73	74	75	76	77	78	79
r9	80	81	82	83	84	85	86	87	88	89
r10	90	91	92	93	94	95	96	97	98	99

Let's apply a condition on column c1, say `c1 > 11`
based on the conditional selection, the out put will be:

In [63]:
```
df['c1'] > 11
#df[df['c1']>11]
```

Out[63]:
```
r1     False
r2     False
r3      True
r4      True
r5      True
r6      True
r7      True
r8      True
r9      True
r10     True
Name: c1, dtype: bool
```

We don't want `r1` and `r2` as they return NaN or null values.
Let's filter the rows based on condition on column values.

In [64]:
```
df[df['c1']>11] # df[boolean_mask]
# We will use such operation frequently in our course.
```

Out[64]:

	c1	c2	c3	c4	c5	c6	c7	c8	c9	c10
r3	20	21	22	23	24	25	26	27	28	29
r4	30	31	32	33	34	35	36	37	38	39
r5	40	41	42	43	44	45	46	47	48	49
r6	50	51	52	53	54	55	56	57	58	59
r7	60	61	62	63	64	65	66	67	68	69
r8	70	71	72	73	74	75	76	77	78	79
r9	80	81	82	83	84	85	86	87	88	89
r10	90	91	92	93	94	95	96	97	98	99

☞ The above, **" `df[df['c1']>11]` "** is a dataframe with applied condition, we can select any col from this dataframe.
For example:

In [65]:
```
result = df[df['c1']>11]
result['c1']
```

Out[65]:
```
r3     20
r4     30
r5     40
r6     50
r7     60
r8     70
r9     80
r10    90
Name: c1, dtype: int32
```

In [66]:
```
# result['r3'] Row can not be accessed
```

We can do the above operations, (filtering and selecting a columns) in a single line (stack commonds).

In [67]:
```
df[df['c1']>11]['c1']
# Could be little confusing for the beginners, but don't worry, we will
# use such operations frequently in the course as well, you will find
# them very handy.
```

Out[67]:
```
r3     20
r4     30
r5     40
r6     50
r7     60
r8     70
r9     80
r10    90
Name: c1, dtype: int32
```

In [68]:
```
# let's split the above operation into its steps to understand
bool_ser = df['c1']>11 # output bool_ser
result = df[bool_ser] # output result
result['c1'] # out put final
```

Out[68]:
```
r3     20
r4     30
r5     40
r6     50
r7     60
r8     70
r9     80
r10    90
Name: c1, dtype: int32
```

In [69]:
```
# let's grab two columns, we need to pass the list ['c1','c9'] here
df[df['c1']>11][['c1','c9']]
```

Out[69]:

	c1	c9
r3	20	28
r4	30	38
r5	40	48
r6	50	58
r7	60	68
r8	70	78
r9	80	88
r10	90	98

In [70]:
```
# We can do this operation on rows using loc
# Passing multiple rows in a list

df[df['c1']>11].loc[['r3','r5']]
```

Out[70]:

	c1	c2	c3	c4	c5	c6	c7	c8	c9	c10
r3	20	21	22	23	24	25	26	27	28	29
r5	40	41	42	43	44	45	46	47	48	49

Let's return a row from our dataframe that have a value 70 in c1

In [71]:
```
df[df['c1']==70]
```

Out[71]:

	c1	c2	c3	c4	c5	c6	c7	c8	c9	c10
r8	70	71	72	73	74	75	76	77	78	79

Combine 2 conditions

Let's try on c1 for a value > 60 and on c2 for a value > 80

In [72]:
```
df[(df['c1']>60) & (df['c2']>80)]
# notice (df['c1']>60)&(df['c2']>80) in () for clear saperation
# with in [] wrapped in df []
```

Out[72]:

	c1	c2	c3	c4	c5	c6	c7	c8	c9	c10
r9	80	81	82	83	84	85	86	87	88	89
r10	90	91	92	93	94	95	96	97	98	99

In [73]:
```
df[(df['c1']>60) and (df['c2']>80)]
```

```
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
<ipython-input-73-5de458536cbe> in <module>
----> 1 df[(df['c1']>60) and (df['c2']>80)]

C:\ProgramData\Anaconda3\lib\site-packages\pandas\core\generic.py in __nonzero_
_(self)
   1476         raise ValueError("The truth value of a {0} is ambiguous. "
   1477                          "Use a.empty, a.bool(), a.item(), a.any() or
 a.all()."
-> 1478                          .format(self.__class__.__name__))
   1479
   1480     __bool__ = __nonzero__

ValueError: The truth value of a Series is ambiguous. Use a.empty, a.bool(), a.
item(), a.any() or a.all().
```

✅**NOTE:**
"and" operator will not work in the above condition and using "and" will return

```
*ValueError: The truth value of a Series is ambiguous. Use a.empty,
a.bool(), a.item(), a.any() or a.all().
```

This "ambiguous" means, True, only work for a single booleans at a time "True and False". We need to use "&" instead. ("|" for or)
Try the above code using "and"
The "and" operator gets confused with series of True/False and raise Error

Let's have a quick look on couple of useful methods.

We will explore more later on in the course!

`reset_index()` and **`set_index()`**
We can reset the index of our dataframe to numerical index (which is default index), `inplace = True` to make the permanent change. *The existing index will be a new column.*

In [74]:
```
1 df.reset_index(inplace = True)
2 df
```

Out[74]:

	index	c1	c2	c3	c4	c5	c6	c7	c8	c9	c10
0	r1	0	1	2	3	4	5	6	7	8	9
1	r2	10	11	12	13	14	15	16	17	18	19
2	r3	20	21	22	23	24	25	26	27	28	29
3	r4	30	31	32	33	34	35	36	37	38	39
4	r5	40	41	42	43	44	45	46	47	48	49
5	r6	50	51	52	53	54	55	56	57	58	59
6	r7	60	61	62	63	64	65	66	67	68	69
7	r8	70	71	72	73	74	75	76	77	78	79
8	r9	80	81	82	83	84	85	86	87	88	89
9	r10	90	91	92	93	94	95	96	97	98	99

** consider, We have a column in our data that could be a useful index,
we want to set that column as an index!**

In [75]:
```
df = pd.DataFrame(data = array_2d, index = index, columns = columns)
newind = 'a b c d e f g h i j'.split() # split at white spaces
# let put newind as a col in the df
#df2 = df
df['newind']=newind
df
#df = pd.DataFrame(data=array_2d, index=index, columns=columns)
```

Out[75]:

	c1	c2	c3	c4	c5	c6	c7	c8	c9	c10	newind
r1	0	1	2	3	4	5	6	7	8	9	a
r2	10	11	12	13	14	15	16	17	18	19	b
r3	20	21	22	23	24	25	26	27	28	29	c
r4	30	31	32	33	34	35	36	37	38	39	d
r5	40	41	42	43	44	45	46	47	48	49	e
r6	50	51	52	53	54	55	56	57	58	59	f
r7	60	61	62	63	64	65	66	67	68	69	g
r8	70	71	72	73	74	75	76	77	78	79	h
r9	80	81	82	83	84	85	86	87	88	89	i
r10	90	91	92	93	94	95	96	97	98	99	j

In [76]:
```
# setting newind as an index, needs to be inplaced
df.set_index('newind', inplace = True)
```

In [77]:
```
df
```

Out[77]:

newind	c1	c2	c3	c4	c5	c6	c7	c8	c9	c10
a	0	1	2	3	4	5	6	7	8	9
b	10	11	12	13	14	15	16	17	18	19
c	20	21	22	23	24	25	26	27	28	29
d	30	31	32	33	34	35	36	37	38	39
e	40	41	42	43	44	45	46	47	48	49
f	50	51	52	53	54	55	56	57	58	59
g	60	61	62	63	64	65	66	67	68	69
h	70	71	72	73	74	75	76	77	78	79
i	80	81	82	83	84	85	86	87	88	89
j	90	91	92	93	94	95	96	97	98	99

head(), tail()

In [78]:
```
# Returns first n rows
df.head(n=2) # n = 5 by default
```

Out[78]:

newind	c1	c2	c3	c4	c5	c6	c7	c8	c9	c10
a	0	1	2	3	4	5	6	7	8	9
b	10	11	12	13	14	15	16	17	18	19

In [79]:
```
# Returns last n rows
df.tail(n=2) # n = 5 by default
```

Out[79]:

newind	c1	c2	c3	c4	c5	c6	c7	c8	c9	c10
i	80	81	82	83	84	85	86	87	88	89
j	90	91	92	93	94	95	96	97	98	99

info()

Provides a concise summary of the DataFrame.

In [80]:
```

df.info()
```

```
<class 'pandas.core.frame.DataFrame'>
Index: 10 entries, a to j
Data columns (total 10 columns):
c1     10 non-null int32
c2     10 non-null int32
c3     10 non-null int32
c4     10 non-null int32
c5     10 non-null int32
c6     10 non-null int32
c7     10 non-null int32
c8     10 non-null int32
c9     10 non-null int32
c10    10 non-null int32
dtypes: int32(10)
memory usage: 480.0+ bytes
```

describe()

Generates descriptive statistics that summarize the central tendency, dispersion and shape of a dataset's distribution, excluding `NaN` values.

In [81]:
```
df.describe()
```

Out[81]:

	c1	c2	c3	c4	c5	c6	c7	c8	
count	10.000000	10.000000	10.000000	10.000000	10.000000	10.000000	10.000000	10.000000	10.
mean	45.000000	46.000000	47.000000	48.000000	49.000000	50.000000	51.000000	52.000000	53.
std	30.276504	30.276504	30.276504	30.276504	30.276504	30.276504	30.276504	30.276504	30.
min	0.000000	1.000000	2.000000	3.000000	4.000000	5.000000	6.000000	7.000000	8.
25%	22.500000	23.500000	24.500000	25.500000	26.500000	27.500000	28.500000	29.500000	30.
50%	45.000000	46.000000	47.000000	48.000000	49.000000	50.000000	51.000000	52.000000	53.
75%	67.500000	68.500000	69.500000	70.500000	71.500000	72.500000	73.500000	74.500000	75.
max	90.000000	91.000000	92.000000	93.000000	94.000000	95.000000	96.000000	97.000000	98.

Hierarchical Indexing

Hierarchical indexing is an important feature of pandas. It makes it possible to have multiple (two or more) index levels on an axis. Somewhat abstractly, it provides a way to work with higher dimensional data in a lower dimensional form.

Let's start with a simple example for **Series**:

In [82]:
```
# Create a Series with a list of lists (or arrays) as the index:
index = [['a','a','a','b','b','b','c','c','d','d'], # level 1 index
         [1,2,3,1,2,3,1,2,1,2]]                      # level 2 index
ser = pd.Series(np.random.randn(10),index = index)  # mean 0 and variance 1.
ser
```

Out[82]:
```
a  1   -0.097603
   2   -1.020513
   3    0.393764
b  1    0.777136
   2    0.054634
   3   -0.850093
c  1   -0.911214
   2   -1.458672
d  1   -0.725856
   2    1.015324
dtype: float64
```

With a hierarchically indexed object, so-called partial indexing is possible, which enables the concise selection of the subsets of the data.

In [83]:
```
# Data retrieval
ser['a']
```

Out[83]:
```
1   -0.097603
2   -1.020513
3    0.393764
dtype: float64
```

In [84]:
```
# single value
ser['a'][2]
```

Out[84]:
```
-1.020513453858637
```

** Example with DataFrame:**
With a DataFrame, either axis can have a hierarchical index.

In [85]:
```
df = pd.DataFrame(np.arange(12).reshape((4, 3)),
                  index=[['a', 'a', 'b', 'b'], [1, 2, 1, 2]],
                  columns=['AB', 'ON', 'BC'])
```

In [86]:
```
df
```

Out[86]:

		AB	ON	BC
a	**1**	0	1	2
	2	3	4	5
b	**1**	6	7	8
	2	9	10	11

How to index the above dataframe!

- on the columns axis, just use normal bracket notation `df[]` .
- on row axis, we use `df.loc[]`

Calling one level of the index returns the sub-dataframe.

In [87]:
```
df['AB']
```

Out[87]:
```
a  1    0
   2    3
b  1    6
   2    9
Name: AB, dtype: int32
```

In [88]:
```
df.loc['a']
```

Out[88]:

	AB	ON	BC
1	0	1	2
2	3	4	5

We want to **grab a single value**, idea is to **go from outside to inside**, e.g. we want to grab "11"

In [89]:
```
#df.loc['b']
#df.loc['b'].loc[2]
print(df.loc['b'].loc[2]['BC'])
df.loc['a'].loc[2]['BC']
```

```
11
```

Out[89]:
```
5
```

In [90]:
```
df.loc['b'].loc[2,'BC']
```

Out[90]:
```
11
```

The hierarchical levels can have names (as strings or any Python objects). If so, these will show up in the console output:

In [91]:
```
df.index.names
```

Out[91]:
```
FrozenList([None, None])
```

Let's give names to the index "L_1, L_2"

In [92]:
```
df.index.names = ['L_1', 'L_2']
```

In [93]:
```
df
```

Out[93]:

		AB	ON	BC
L_1	**L_2**			
a	**1**	0	1	2
	2	3	4	5
b	**1**	6	7	8
	2	9	10	11

Good to know!

xs()

Let me introduce a very useful and built-in method " `xs()` " to grab data from multilevel index.
`xs()` has ability to go inside a multilevel index.

In [94]:
```
# Returns a cross-section (row(s) or column(s)) from the Series/DataFrame.
df.xs('a')
```

Out[94]:

L_2	AB	ON	BC
1	0	1	2
2	3	4	5

If we want to grab all the data in df where index L_2 is "1", its tricky for `loc` method, `xs` will do the magic here!
For Example:
tell `xs()` what you want, 1 here, and indicate the level, L_2 in this case.z

In [95]:
```
df.xs(1, level='L_2')
```

Out[95]:

L_1	AB	ON	BC
a	0	1	2
b	6	7	8

Hi Guys,
Welcome back to the pandas essentials, now we are going to talk about the missing data!

Handling Missing Data

Missing data is very common in many data analysis applications. pandas has a great ability to deal with the missing data.
Let's learn some convenient methods to deal with **missing data in pandas**:

In [96]:
```
import numpy as np
import pandas as pd
```

Creating ad dataframe with missing data

In [97]:
```
data_dic = {'A':[1,2,np.nan,4,np.nan],
            'B':[np.nan,np.nan,np.nan,np.nan,np.nan],
            'C':[11,12,13,14,15],
            'D':[16,np.nan,18,19,20]}
df = pd.DataFrame(data_dic) # dataframe from a dic.
```

In [98]:
```
df
```

Out[98]:

	A	B	C	D
0	1.0	NaN	11	16.0
1	2.0	NaN	12	NaN
2	NaN	NaN	13	18.0
3	4.0	NaN	14	19.0
4	NaN	NaN	15	20.0

isnull(), notnull() -- Check for missing data in the dataset!

In [99]:
```
# isnull() returns True if the data is missing
df.isnull()
```

Out[99]:

	A	B	C	D
0	False	True	False	False
1	False	True	False	True
2	True	True	False	False
3	False	True	False	False
4	True	True	False	False

In [100]:
```
df.isnull().sum(axis=1)
```

Out[100]:
```
0    1
1    2
2    2
3    1
4    2
dtype: int64
```

In [101]:
```
# notnull() returns True for non-NaN values
df.notnull()
```

Out[101]:

	A	B	C	D
0	True	False	True	True
1	True	False	True	False
2	False	False	True	True
3	True	False	True	True
4	False	False	True	True

☞ NaN as "0" for sum()

In [102]:
```
# Sum on Column "A", (NaN as 0)
df['A'].sum()
```

Out[102]:
```
7.0
```

☞ NaN ignored for mean(). Hence that row or column will not be considered.

In [103]:
```
df
```

Out[103]:

	A	B	C	D
0	1.0	NaN	11	16.0
1	2.0	NaN	12	NaN
2	NaN	NaN	13	18.0
3	4.0	NaN	14	19.0
4	NaN	NaN	15	20.0

In [104]:
```
df.mean()
```

Out[104]:
```
A     2.333333
B          NaN
C    13.000000
D    18.250000
dtype: float64
```

In [105]:
```
df['C'].mean()
```

Out[105]:
```
13.0
```

dropna(), fillna() -- Cleaning / filling the missing data

In [106]:
```
# drop any row (dafault value) with any NaN value
df.dropna()
```

Out[106]:

	A	B	C	D

In [107]:
```
# for column, need to tell axis = 1
df.dropna(axis=1)
```

Out[107]:

	C
0	11
1	12
2	13
3	14
4	15

thresh : int, default None thresh = 3 means, it will drop any column that have less than 3 non-NaN values. OR a column has at least 3 non-NaNs to survive.

In [108]:
```
1 df
```

Out[108]:

	A	B	C	D
0	1.0	NaN	11	16.0
1	2.0	NaN	12	NaN
2	NaN	NaN	13	18.0
3	4.0	NaN	14	19.0
4	NaN	NaN	15	20.0

In [109]:
```
1 df.dropna(thresh=3, axis=1)
```

Out[109]:

	A	C	D
0	1.0	11	16.0
1	2.0	12	NaN
2	NaN	13	18.0
3	4.0	14	19.0
4	NaN	15	20.0

We can use fillna() to fill in the values.
inplaced = True for permanent change.

In [110]:
```
1 df.fillna(value='Filled')
```

Out[110]:

	A	B	C	D
0	1	Filled	11	16
1	2	Filled	12	Filled
2	Filled	Filled	13	18
3	4	Filled	14	19
4	Filled	Filled	15	20

Let's fill in the values using mean of the column.

In [111]:
```
df['A'].fillna(value = df['A'].mean())
```

Out[111]:
```
0    1.000000
1    2.000000
2    2.333333
3    4.000000
4    2.333333
Name: A, dtype: float64
```

In [112]:
```
# pad / ffill: Forward fill, last valid observation forward to next NaN
df.fillna(method='ffill')

```

Out[112]:

	A	B	C	D
0	1.0	NaN	11	16.0
1	2.0	NaN	12	16.0
2	2.0	NaN	13	18.0
3	4.0	NaN	14	19.0
4	4.0	NaN	15	20.0

In [113]:
```
print(df)
df.fillna(method='pad')
```

```
     A   B   C     D
0  1.0 NaN  11  16.0
1  2.0 NaN  12   NaN
2  NaN NaN  13  18.0
3  4.0 NaN  14  19.0
4  NaN NaN  15  20.0
```

Out[113]:

	A	B	C	D
0	1.0	NaN	11	16.0
1	2.0	NaN	12	16.0
2	2.0	NaN	13	18.0
3	4.0	NaN	14	19.0
4	4.0	NaN	15	20.0

In [114]:
```
#bfill/backfill -- use NEXT valid observation to fill gap
df.fillna(method='bfill')
```

Out[114]:

	A	B	C	D
0	1.0	NaN	11	16.0
1	2.0	NaN	12	18.0
2	4.0	NaN	13	18.0
3	4.0	NaN	14	19.0
4	NaN	NaN	15	20.0

In [115]:
```
# fill with you own given value
df.fillna(0)
```

Out[115]:

	A	B	C	D
0	1.0	0.0	11	16.0
1	2.0	0.0	12	0.0
2	0.0	0.0	13	18.0
3	4.0	0.0	14	19.0
4	0.0	0.0	15	20.0

Combining and Merging Datasets

Data contained in pandas objects can be combined together in a number of ways:

- **`merge()`:** connects rows in DataFrames based on one or more keys. (*This will be familiar to SQL or other relational databases users, as it implements database join operations*).
- **`concat()`:** concatenate or "stacks" together objects along an axis.

✅ If you don't know SQL, don't worry, the concepts of merging are presented with very simple examples so that you can follow the steps. Although, our focus here is not to learn SQL, we only want to go through the widely used and few very important **`inner`** and **`outer`** joining operations for data wrangling.
If you have questions, please ask and we are more than happy to help!
✅ Important thing you should know: Merging operation may give **`NaN`** in the output and they needs to be treated according to the circumstances/requirements during data analysis.

Let's discuss these methods with examples.

Database-Style DataFrame joins

Merge or join operations combine datasets by linking rows using one or more keys. These operations are central to relational databases (e.g., SQL-based).

In [116]:
```
## import pandas as pd
```

We need data to work with, let's create two DataFrames, df1 and df2.

In [117]:
```
df1 = pd.DataFrame({'key': ['a', 'b', 'c', 'd', 'e'],'A1': range(5), 'B1':ra
df2 = pd.DataFrame({'key': ['a', 'b', 'c'], 'A2': range(3), 'B2':range(3,6)}
```

Always good to see how our data look like!

In [118]:
```
df1
```

Out[118]:

	key	A1	B1
0	a	0	5
1	b	1	6
2	c	2	7
3	d	3	8
4	e	4	9

In [119]:
```
df2
```

Out[119]:

	key	A2	B2
0	a	0	3
1	b	1	4
2	c	2	5

We have created dataframes, they look great.
Before we move on, let's explore **'merge()'** method first.
We can type pd.merge and press shift+tab in the Jupyter notebook to see the documentation.
There are several parameters that we can pass to the merge method, the most important ones are **'how'** and **'on'**, that we will discuss here.

- **'how'** tells the **'merge()'**, what type of joining operation needs to be done, it could be **'inner','outer','left','right'**. Default value of **'how'** is **'inner**, if nothing is provided.
- **'on'** tells the field name to join on, which could be a label or a list.

merge()

Let's overview **'how'** and **'on'** parameters in **'merge()'**.

how:{'inner','outer','left','right'}

- **'inner':** use intersection of keys from both frames, similar to a SQL inner join.
- **'outer':** use union of keys from both frames, similar to a SQL full outer join.

- `'left'`: use only keys from left frame, similar to a SQL left outer join.
- `'right'`: use only keys from right frame, similar to a SQL right outer join.

on:label or list

- Field names to join on.
- Must be found in both DataFrames.

how ='inner'

The key column in the resultant will be the intersection of the 'key' columns in both df1 and df2. In our case, a b c along with the associated data

☞ I am using `print` to out put resultant along with the original dataframes `df1` , `df2` to do the comparisons.

In [120]:
```
print(pd.merge(df1, df2, how = 'inner', on='key'))
print(df1)
print(df2)
```

```
  key  A1  B1  A2  B2
0   a   0   5   0   3
1   b   1   6   1   4
2   c   2   7   2   5
  key  A1  B1
0   a   0   5
1   b   1   6
2   c   2   7
3   d   3   8
4   e   4   9
  key  A2  B2
0   a   0   3
1   b   1   4
2   c   2   5
```

☞ d, e did not appear in the merged output, `'inner'` returns the intersection of key columns only!

how = 'Outer'

- The key column in the result will be the union of `df1['key']` and `df2['key']` , means, all the keys found in both tables.

☞ I am using `print` to out put resultant along with the original dataframes `df1` , `df2` to do the comparisons.

In [121]:
```
print(pd.merge(df1, df2, how = 'outer', on='key'))
print(df1)
print(df2)
```

```
  key  A1  B1   A2   B2
0   a   0   5  0.0  3.0
1   b   1   6  1.0  4.0
2   c   2   7  2.0  5.0
3   d   3   8  NaN  NaN
4   e   4   9  NaN  NaN
  key  A1  B1
0   a   0   5
1   b   1   6
2   c   2   7
3   d   3   8
4   e   4   9
  key  A2  B2
0   a   0   3
1   b   1   4
2   c   2   5
```

☞ `NaN` in A2, B2 columns for d, e indexes. Its Union operation and A2, B2 values does not exist in df2 for indexes d, e!

how ='left'

Use only key column of the left dataframe, similar to a SQL left outer join.

In [122]:
```
print(pd.merge(df1, df2, how ='left',on='key'))
print(df1)
print(df2)
```

```
  key  A1  B1   A2   B2
0   a   0   5  0.0  3.0
1   b   1   6  1.0  4.0
2   c   2   7  2.0  5.0
3   d   3   8  NaN  NaN
4   e   4   9  NaN  NaN
  key  A1  B1
0   a   0   5
1   b   1   6
2   c   2   7
3   d   3   8
4   e   4   9
  key  A2  B2
0   a   0   3
1   b   1   4
2   c   2   5
```

☞ `NaN` for indexes d, e in A2, B2, as indexes d, e don't exist in `df2['key']` .

how = 'right'

Use only key column of the right dataframe, similar to a SQL right outer join.

In [123]:
```
print(pd.merge(df1, df2, how = 'right',on='key'))
print(df1)
print(df2)
```

```
  key  A1  B1  A2  B2
0   a   0   5   0   3
1   b   1   6   1   4
2   c   2   7   2   5
  key  A1  B1
0   a   0   5
1   b   1   6
2   c   2   7
3   d   3   8
4   e   4   9
  key  A2  B2
0   a   0   3
1   b   1   4
2   c   2   5
```

Merging example with two key (key1, key2) columns -- little complicated!

Let's create two data frames such that each have two key columns, `key1 & key2` .

In [124]:
```
left = pd.DataFrame({'key1': ['a', 'a', 'b', 'c'],
                     'key2': ['a', 'b', 'a', 'b'],
                     'A': ['A0', 'A1', 'A2', 'A3'],
                     'B': ['B0', 'B1', 'B2', 'B3']})

right = pd.DataFrame({'key1': ['a', 'b', 'b', 'c'],
                      'key2': ['a', 'b', 'a', 'a'],
                      'C': ['C0', 'C1', 'C2', 'C3'],
                      'D': ['D0', 'D1', 'D2', 'D3']})
```

`'inner'` is intersection, only the key pair present in both dataframes will appear in the resultant

In [125]:
```
print(left)
print(right)
print(pd.merge(left, right, how = 'inner', on=['key1', 'key2']))
```

```
  key1 key2   A   B
0    a    a  A0  B0
1    a    b  A1  B1
2    b    a  A2  B2
3    c    b  A3  B3
  key1 key2   C   D
0    a    a  C0  D0
1    b    b  C1  D1
2    b    a  C2  D2
3    c    a  C3  D3
  key1 key2   A   B   C   D
0    a    a  A0  B0  C0  D0
1    b    a  A2  B2  C2  D2
```

As we know, **`'outer'`** is union, all key pair present in both dataframes will appear in the resultant.

In [126]:
```
print(left)
print(right)
print(pd.merge(left, right, how='outer', on=['key1', 'key2']))
```

```
  key1 key2   A   B
0    a    a  A0  B0
1    a    b  A1  B1
2    b    a  A2  B2
3    c    b  A3  B3
  key1 key2   C   D
0    a    a  C0  D0
1    b    b  C1  D1
2    b    a  C2  D2
3    c    a  C3  D3
  key1 key2    A    B    C    D
0    a    a   A0   B0   C0   D0
1    a    b   A1   B1  NaN  NaN
2    b    a   A2   B2   C2   D2
3    c    b   A3   B3  NaN  NaN
4    b    b  NaN  NaN   C1   D1
5    c    a  NaN  NaN   C3   D3
```

For **`'left'`** join, the key pair in left will be used only

In [127]:
```
print(left)
print(right)
print(pd.merge(left, right, how='left', on=['key1', 'key2']))
```

```
  key1 key2   A   B
0    a    a  A0  B0
1    a    b  A1  B1
2    b    a  A2  B2
3    c    b  A3  B3
  key1 key2   C   D
0    a    a  C0  D0
1    b    b  C1  D1
2    b    a  C2  D2
3    c    a  C3  D3
  key1 key2   A   B    C    D
0    a    a  A0  B0   C0   D0
1    a    b  A1  B1  NaN  NaN
2    b    a  A2  B2   C2   D2
3    c    b  A3  B3  NaN  NaN
```

For **'right'** join, the key pair in right will be used only

In [128]:
```
print(left)
print(right)
print(pd.merge(left, right, how='right', on=['key1', 'key2']))
```

```
  key1 key2   A   B
0    a    a  A0  B0
1    a    b  A1  B1
2    b    a  A2  B2
3    c    b  A3  B3
  key1 key2   C   D
0    a    a  C0  D0
1    b    b  C1  D1
2    b    a  C2  D2
3    c    a  C3  D3
  key1 key2    A    B   C   D
0    a    a   A0   B0  C0  D0
1    b    a   A2   B2  C2  D2
2    b    b  NaN  NaN  C1  D1
3    c    a  NaN  NaN  C3  D3
```

Concatenation

Concatenation is interchangeably referred as binding, or stacking as well. This operation basically glues together DataFrames.

☞ It's important to remember that dimensions should match along the axis, we are concatenating on.

We can use **pd.concat** and pass in a list of DataFrames to concatenate together.
Let's create two simple dataframes, with the given indexes, to understand concatenation.

In [129]:
```
df1 = pd.DataFrame({'A': ['A0', 'A1', 'A2', 'A3'],
                    'B': ['B0', 'B1', 'B2', 'B3'],
                    'C': ['C0', 'C1', 'C2', 'C3'],
                    'D': ['D0', 'D1', 'D2', 'D3']},
                   index=[0, 1, 2, 3])
```

In [130]:
```
df2 = pd.DataFrame({'A': ['A4', 'A5', 'A6', 'A7'],
                    'B': ['B4', 'B5', 'B6', 'B7'],
                    'C': ['C4', 'C5', 'C6', 'C7'],
                    'D': ['D4', 'D5', 'D6', 'D7']},
                    index=[4,5,6,7])
```

In [131]:
```
df1
```

Out[131]:

	A	B	C	D
0	A0	B0	C0	D0
1	A1	B1	C1	D1
2	A2	B2	C2	D2
3	A3	B3	C3	D3

In [132]:
```
df2
```

Out[132]:

	A	B	C	D
4	A4	B4	C4	D4
5	A5	B5	C5	D5
6	A6	B6	C6	D6
7	A7	B7	C7	D7

In [133]:
```
pd.concat([df1,df2]) # default axis is 0/'index' to concatenate along
```

Out[133]:

	A	B	C	D
0	A0	B0	C0	D0
1	A1	B1	C1	D1
2	A2	B2	C2	D2
3	A3	B3	C3	D3
4	A4	B4	C4	D4
5	A5	B5	C5	D5
6	A6	B6	C6	D6
7	A7	B7	C7	D7

In [134]:
```
1 pd.concat([df1,df2],axis=1) # axis = 1/columns
```

Out[134]:

	A	B	C	D	A	B	C	D
0	A0	B0	C0	D0	NaN	NaN	NaN	NaN
1	A1	B1	C1	D1	NaN	NaN	NaN	NaN
2	A2	B2	C2	D2	NaN	NaN	NaN	NaN
3	A3	B3	C3	D3	NaN	NaN	NaN	NaN
4	NaN	NaN	NaN	NaN	A4	B4	C4	D4
5	NaN	NaN	NaN	NaN	A5	B5	C5	D5
6	NaN	NaN	NaN	NaN	A6	B6	C6	D6
7	NaN	NaN	NaN	NaN	A7	B7	C7	D7

Good to know! -- (Optional)

Joining

Joining is a convenient method for combining the columns of two potentially differently-indexed DataFrames into a single result DataFrame.

In [135]:
```
1 left = pd.DataFrame({'A': ['A0', 'A1', 'A2'],
2                      'B': ['B0', 'B1', 'B2']},
3                       index=['K0', 'K1', 'K2'])
4
5 right = pd.DataFrame({'C': ['C0', 'C2', 'C3'],
6                       'D': ['D0', 'D2', 'D3']},
7                       index=['K0', 'K2', 'K3'])
```

In [136]:
```
1 left
```

Out[136]:

	A	B
K0	A0	B0
K1	A1	B1
K2	A2	B2

In [137]:
```
1 right
```

Out[137]:

	C	D
K0	C0	D0
K2	C2	D2
K3	C3	D3

In [138]:
```
1 left.join(right)
```

Out[138]:

	A	B	C	D
K0	A0	B0	C0	D0
K1	A1	B1	NaN	NaN
K2	A2	B2	C2	D2

Groupby

Groupby is one of the most important and key functionality in pandas. It allows us to group data together, call aggregate functions and combine the results in three steps *split-apply-combine*: Before we move on to the hands-on, let's try to understand how this split-apply-combine work, using a data in different colours!

- **Split:** In this process, data contained in a pandas object (e.g. Series, DataFrame) is split into groups based on one or more keys that we provide. The splitting is performed on a particular axis of an object. For example, a DataFrame can be grouped on its rows (axis=0) or its columns (axis=1).
- **apply:** Once splitting is done, a function is applied to all groups independently, producing a new value.
- **combine:** Finally, the results of all those functions applications are combined into a resultant object. The form of the resulting object will usually depend on what's being done to the data.

Lets explore with some examples:

In [139]:
```
1 # import pandas as pd
```

Let's create a dictionary and convert that into pandas dataframe

In [140]:
```
# Create a dataframe
data = {'Store':['Walmart','Walmart','Costco','Costco','Target','Target'],
        'Customer':['Tim','Jermy','Mark','Denice','Ray','Sam'],
        'Sales':[150,200,550,90,430,120]}
df = pd.DataFrame(data)
df
```

Out[140]:

	Store	Customer	Sales
0	Walmart	Tim	150
1	Walmart	Jermy	200
2	Costco	Mark	550
3	Costco	Denice	90
4	Target	Ray	430
5	Target	Sam	120

In the df, we have a Customer unique name, Sales in numbers and store name.
Let's group the data, in df, based on column "Store" using groupby method. This will create a DataFrameGroupBy object.

Grab the df, access the gropby method using "." and pass the column we want to group the data on.
Notice, we get a groupby object, stored in a memory 0x....

In [141]:
```
df.groupby("Store")
```

Out[141]:
```
<pandas.core.groupby.generic.DataFrameGroupBy object at 0x0000024A4F9BBBE0>
```

Let's save the created object as a new variable.

In [142]:
```
by_store = df.groupby("Store")
```

Now, we have grouped data in "by_store" object, we can call aggregate method on this object.

In [143]:
```
by_store.mean()
```

Out[143]:

	Sales
Store	
Costco	320
Target	275
Walmart	175

Pandas will apply `mean()` on number columns "Sales". It ignore not numeric columns automatically. Same is True for sum, std, max, and so on..

In [144]:
```
# The steps above in a sinlge line code
df.groupby('Store').mean()
```

Out[144]:

Store	Sales
Costco	320
Target	275
Walmart	175

Notice that, the result is a dataframe with "Store" as index and "Sales" as column. We can use loc method to locate any value for certain company after aggregation function. This will give us the value (e.g. sales) for a single store.

In [145]:
```
# In oneline code
df.groupby('Store').sum().loc["Target"]
```

Out[145]:
```
Sales    550
Name: Target, dtype: int64
```

We can perform whole lots of aggregation operations on "by_store" object.

In [146]:
```
by_store.min()
```

Out[146]:

Store	Customer	Sales
Costco	Denice	90
Target	Ray	120
Walmart	Jermy	150

In [147]:
```
by_store.max()
```

Out[147]:

Store	Customer	Sales
Costco	Mark	550
Target	Sam	430
Walmart	Tim	200

In [148]:
```
1 by_store.std()
```

Out[148]:

Store	Sales
Costco	325.269119
Target	219.203102
Walmart	35.355339

In [149]:
```
1 # count the no of instances in the columns, works with strings as well
2 # we have 2 customers and 2 sales in each store
3 by_store.count()
```

Out[149]:

Store	Customer	Sales
Costco	2	2
Target	2	2
Walmart	2	2

describe is a useful method, that gives a bunch of useful information, such as, mean, min, quartile values etc for each company.

In [150]:
```
1 by_store.describe()
```

Out[150]:

	Sales							
	count	mean	std	min	25%	50%	75%	max
Store								
Costco	2.0	320.0	325.269119	90.0	205.0	320.0	435.0	550.0
Target	2.0	275.0	219.203102	120.0	197.5	275.0	352.5	430.0
Walmart	2.0	175.0	35.355339	150.0	162.5	175.0	187.5	200.0

Let's use `transpose()` after describe so that the output looks good!

In [151]:
```
1 by_store.describe().transpose()
```

Out[151]:

	Store	Costco	Target	Walmart
Sales	**count**	2.000000	2.000000	2.000000
	mean	320.000000	275.000000	175.000000
	std	325.269119	219.203102	35.355339
	min	90.000000	120.000000	150.000000
	25%	205.000000	197.500000	162.500000
	50%	320.000000	275.000000	175.000000
	75%	435.000000	352.500000	187.500000
	max	550.000000	430.000000	200.000000

We can call a column name for a selected store to separate information with `transpose()` as well!

In [152]:
```
1 by_store.describe().transpose()['Costco']
```

Out[152]:
```
Sales  count      2.000000
       mean     320.000000
       std      325.269119
       min       90.000000
       25%      205.000000
       50%      320.000000
       75%      435.000000
       max      550.000000
Name: Costco, dtype: float64
```

Useful methods and operations

There are lots of options available in pandas to explore and get the basic statistics on your data. We have already covered some of them e.g. `head()`, `isnull()`, `dropna()`, `fillna()` etc.

In this lecture, we will explore some more general purpose operations and revise what we have learned in the previous lectures.
Let's create a dataframe to get hands-on experience on these operations.
I will repeat some values and also generate NaN in our dataframe.

In [153]:
```
# import numpy as np
# import pandas as pd
data_dic = {'col_1':[1,2,3,4,5],
            'col_2':[111,222,333,111,555],
            'col_3':['alpha','bravo','charlie',np.nan,np.nan],
            }
df = pd.DataFrame(data_dic,index=[1,2,3,4,5])
df
```

Out[153]:

	col_1	col_2	col_3
1	1	111	alpha
2	2	222	bravo
3	3	333	charlie
4	4	111	NaN
5	5	555	NaN

Lets start with what we know.

info()

provides a concise summary of a DataFrame. We will use this function very often in the course.

In [154]:
```
df.info()
```

```
<class 'pandas.core.frame.DataFrame'>
Int64Index: 5 entries, 1 to 5
Data columns (total 3 columns):
col_1    5 non-null int64
col_2    5 non-null int64
col_3    3 non-null object
dtypes: int64(2), object(1)
memory usage: 160.0+ bytes
```

head(n)

Returns the first n rows, default is 5. This is very useful to get the overview on our data. We will use this very often in the course.

In [155]:
```
df.head(2)
```

Out[155]:

	col_1	col_2	col_3
1	1	111	alpha
2	2	222	bravo

isnull()

Return a boolean same-sized object indicating if the values are null.

In [156]:
```
1 df.isnull()
```

Out[156]:

	col_1	col_2	col_3
1	False	False	False
2	False	False	False
3	False	False	False
4	False	False	True
5	False	False	True

dropna()

- `axis = 0/rows, 1/columns` -- 0 is default
- `inplace = False` by default, to make the permanent change, needs to be True

using print function to compare the output for both axis

In [157]:
```
1 print(df.dropna(axis = 0))
2 print(df.dropna(axis = 1))
```

```
   col_1  col_2    col_3
1      1    111    alpha
2      2    222    bravo
3      3    333  charlie
   col_1  col_2
1      1    111
2      2    222
3      3    333
4      4    111
5      5    555
```

fillna()

Fill NA/NaN values using the specified method

- `value = None` by default
- `method = None` by default ('backfill', 'ffill' etc)
- `axis = 0/row or index, 1/columns`
- `inplace = False` by default, If `True` , fill in place and the data will be modified.

In [158]:
```
# df.fillna() # ValueError: must specify a fill method or value
print(df.fillna(value = 'XYZ'))
print(df.fillna(method = 'ffill'))
```

```
   col_1  col_2    col_3
1      1    111    alpha
2      2    222    bravo
3      3    333  charlie
4      4    111      XYZ
5      5    555      XYZ
   col_1  col_2    col_3
1      1    111    alpha
2      2    222    bravo
3      3    333  charlie
4      4    111  charlie
5      5    555  charlie
```

unique()

Find and returns all the unique values.
Lets see how it works on all the columns in our dataframe.

In [159]:
```
print(df['col_1'].unique())
print(df['col_2'].unique())
print(df['col_3'].unique())
# 111 and NaN are repeated values, unique will only return once.
```

```
[1 2 3 4 5]
[111 222 333 555]
['alpha' 'bravo' 'charlie' nan]
```

nunique()

Find returns "how many unique values exist".
☞ Notice the difference, for NaN, it count a missing value and returns "3" for col_3.

In [160]:
```
print(df['col_1'].nunique())
print(df['col_2'].nunique())
print(df['col_3'].nunique())
```

```
5
4
3
```

value_counts()

We want a table with all the values along with no. of times they appeared in our data, value_counts do the work here!
☞ for NaN, it count a missing value, nothing in the output.

In [161]:
```
print(df['col_1'].value_counts())
print(df['col_2'].value_counts())
print(df['col_3'].value_counts())
```

```
5    1
4    1
3    1
2    1
1    1
Name: col_1, dtype: int64
111    2
222    1
333    1
555    1
Name: col_2, dtype: int64
alpha      1
charlie    1
bravo      1
Name: col_3, dtype: int64
```

✅ **`unique(), unique(), value_counts()`** are three very useful and frequently used methods, which are associated with finding unique values in the data.

`sort_values()`

by default:

- `ascending=True`
- `inplace=False`

In [162]:
```
df.sort_values(by='col_2')
```

Out[162]:

	col_1	col_2	col_3
1	1	111	alpha
4	4	111	NaN
2	2	222	bravo
3	3	333	charlie
5	5	555	NaN

Data Selection

Lets talk about ***Selecting Data*** once again. We have learned to grab data in our previous lectures as well.

- We can grab a column with its name, do the conditional selection and much more ....
- We can use loc and iloc to find rows as well.

Let's revise the conditional selection, this also includes data selection based on the column name.

Lets do the following steps:

```
* df['col_1'] > 2 : returns the data where condition is True (if you rem
ember, this is just a boolean series)
* df['col_2'] == 111 : returns the data where condition is True
* Lets combine these tow conditions with & by putting both conditions in
().
* wrap them in df[] and see what it returns!
```

Our one line code is **`(df['col_1'] > 2) & (df['col_2'] == 111)`**

In [163]:
```
df['col_1'] > 2 # boolean series
```

Out[163]:
```
1    False
2    False
3     True
4     True
5     True
Name: col_1, dtype: bool
```

In [164]:
```
df['col_2'] # boolean series
```

Out[164]:
```
1    111
2    222
3    333
4    111
5    555
Name: col_2, dtype: int64
```

In [165]:
```
"""We can say, this is a boolean mask on said condition to provide
to the dataframe, df, for filtering out the results."""
bool_ser = (df['col_1'] > 2) & (df['col_2'] == 111)
bool_ser
```

Out[165]:
```
1    False
2    False
3    False
4     True
5    False
dtype: bool
```

In [166]:
```
result = df[bool_ser]
result
# df[(df['col_1'] > 2) & (df['col_2'] == 111)]
# In the output below, we got the date based on our provided conditions!
```

Out[166]:

	col_1	col_2	col_3
4	4	111	NaN

apply()

Indeed, this is one of the most powerful pandas feature. Using **`apply()`** method, we can **broadcast** our **customized functions** on our data.
Let's see how to calculate square of col_1

In [167]:
```
# Our customized function to calculate the squares
def square(value):
    return value*2
```

- Let's broadcast our customized function `"square"` using `"apply"` method to calculate squares of the col_1 in our DataFrame, df.

In [168]:
```
df['col_1'].apply(square)
```

Out[168]:
```
1     2
2     4
3     6
4     8
5    10
Name: col_1, dtype: int64
```

- The same operation can be conveniently carried out using state of the art **`lambda`** expression!

In [169]:
```
df['col_1'].apply(lambda value:value*2)
```

Out[169]:
```
1     2
2     4
3     6
4     8
5    10
Name: col_1, dtype: int64
```

In [170]:
```
# Yes, we can use built-in functions with apply as well
# Finding a lenght of strings in the column
df['col_3'][0:3].apply(len)
```

Out[170]:
```
1    5
2    5
3    7
Name: col_3, dtype: int64
```

☞ We avoiding `NaN` in col_3, because:
`TypeError: object of type 'float' has no len()`

In [171]:
```
# Let's confirm the type of NaN
type(np.nan)
```

Out[171]:
```
float
```

Good to know

In [172]:
```
# Getting index names
df.index
```

Out[172]:
```
Int64Index([1, 2, 3, 4, 5], dtype='int64')
```

In [173]:
```
# Getting column names
df.columns
```

Out[173]:
```
Index(['col_1', 'col_2', 'col_3'], dtype='object')
```

In [174]:
```
# Deleting row (axis=0) or column (axis=1)
print(df.drop('col_1',axis=1))
print(df) # inplace = True for permanent change
```

```
   col_2    col_3
1    111    alpha
2    222    bravo
3    333  charlie
4    111      NaN
5    555      NaN
   col_1  col_2    col_3
1      1    111    alpha
2      2    222    bravo
3      3    333  charlie
4      4    111      NaN
5      5    555      NaN
```

In [175]:
```
# deleting col_1 permanently
newdf= df.copy() # creating a copy, may need to use df at later stage
del newdf['col_1']
newdf
```

Out[175]:

	col_2	col_3
1	111	alpha
2	222	bravo
3	333	charlie
4	111	NaN
5	555	NaN

In [176]:
```
df.index
```

Out[176]:
```
Int64Index([1, 2, 3, 4, 5], dtype='int64')
```

pivot_table()

`shift + tab` to read the documentation.

Create a spreadsheet-style pivot table as a DataFrame. The levels in the pivot table will be stored in MultiIndex objects (hierarchical indexes) on the index and columns of the result DataFrame.

`pivot_table` takes three main arguments:

- `values` default is None
- `index` default is None
- `columns` default is None

Let's create a pivot table from our dataframe df.

- We want our data points to be col_2, so, **values = 'col_2'**
- We want our index to be col_1, so, **index = 'col_1'**
- Finally, We want our columns to be defined by col_3, so, **columns = ['col_3']**

☞ If you are an excel user, you may be familiar with pivot_table. If not, don't worry about this at this stage, we will discuss it in the coming sections of the course.

In [177]:
```
df
```

Out[177]:

	col_1	col_2	col_3
1	1	111	alpha
2	2	222	bravo
3	3	333	charlie
4	4	111	NaN
5	5	555	NaN

In [178]:
```
df.pivot_table(values = 'col_2',  index='col_1', columns=['col_3'])
```

Out[178]:

col_3	alpha	bravo	charlie
col_1			
1	111.0	NaN	NaN
2	NaN	222.0	NaN
3	NaN	NaN	333.0

NaN appeared for missing data.
NaN in col_3 will not be used for the column name in the pivot table, skipped index 4 and 5.

Let's have a look on another example for Pivot_table

In [179]:
```
# Creating DataFrame
data = {'A':['foo','foo','foo','bar','bar','bar'],
     'B':['one','one','two','two','one','one'],
       'C':['x','y','x','y','x','y'],
       'D':[1,3,2,5,4,1]}

foobar = pd.DataFrame(data)
```

In [180]:
```
# Our dataframe looks like
foobar
```

Out[180]:

	A	B	C	D
0	foo	one	x	1
1	foo	one	y	3
2	foo	two	x	2
3	bar	two	y	5
4	bar	one	x	4
5	bar	one	y	1

Let's create a pivot table from our dataframe foobar.

- We want our data points to be D, so, **values = 'D'**
- We want our index to be A,B in multilevel index, so, **index = ['A','B']**
- Finally, We want our columns to be defined by C, so, **columns = ['C']**

In [181]:
```
foobar.pivot_table(values='D',index=['A', 'B'],columns=['C'])
```

Out[181]:

		C	x	y
A	**B**			
bar	**one**	4.0	1.0
	two	NaN	5.0
foo	**one**	1.0	3.0
	two	2.0	NaN